**The Introduction**

*This is an introductory curriculum for teachers, it is designed to get a teacher with a bit of familiarity with computers to the point where they can teach the rest of the curriculum. At the end are suggested student projects. This document mirrors the curriculum for students and both can be given to the students. The student worksheets should be given to the students who will follow along as the teacher demonstrates the worked examples, and then breaks to allow the students to work on the exercises.*

**What is programing**

For the purposes of this class, programing is in some sense simply a way of telling your computer to do things for you. The difference between programing and, say, clicking on a link with your mouse, is that programing allows you to tell your computer to do many more things simply display web pages. But at the end of the day the goal is the same, get your computer to do something for your, whether it’s check your email, perform a computation or interact with like a video game.

There are three basic parts of programing:

1. **Variables** store data, like numbers, letters, images, matrices, etc.

2. **Functions** tell your computer to do things, for example add two numbers or display a picture on the screen

3. **Control Structures:** these come in two major flavors, logical (if *x* > 2 then compute *x*2) and loops (do this thing many times).

Those three structures in various combinations make up all of what makes your computer run. Today we’re going to talk about all of them.

**MATLAB**

MATLAB (the MATrix LABoratory) is a programing language that is optimized to perform matrix calculations. For our purposes, matrices will primary be a way to store data.

**Variables and Lists**

Simple arithmetic computations are the easiest thing to do in MATLAB. For example addition, multiplication, division, and powers are the standard symbols. For example if you type

>> 4-(3(10+6))∧(20/6)

into the command line and press “enter” a few things happen: First MATLAB prints

ans = -4.0191e+05

it also adds a variable called ans to the ‘Workspace.’ If you look over at the workspace table in the top right of the page it will have a variable ans with a value -4.0191e+05. The variable ans always contains the result of the last computation.

If you want to save the result of a computation for longer you can assign a variable to it and then use it to do computations::

>> x = 5/3

Storing single variables is all well and good, but where MATLAB really shines is in dealing with lists and vectors. There are sevral ways to make vectors, the easiest is just to type them:

>> x = [1,-3,4,2,-7,8,9]

This will store the vector in the variable x. Making a vector of all the numbers from 3:10 say is also easy, just type

>> x = 3:10

What if we want to make a vector of the numbers between -2 and 2 with a step size of .1? Then we type

>> x = -2:.1:2

This tells us to set x equal to the vector where we start at -2 and count up by .1 until we hit 2. I’ve suppressed the output here because it’s long. You can build almost all the vectors you want this way. Other useful vector creation functions include:

>> x = zeros(i,j)

creates an *i* × *j* matrix of zeros creates an *i* × *j* matrix of ones creates an *i* × *j* matrix of random numbers between 0 and 1

Vectors can have constants added to each element of them, or can be multiplied by a constant:

>> 5+[1,2,3]

Vectors can also be added, subtracted and multiplied together component-wise, provided they have the same length, but you have to remember to add a period “**.**” when doing component wise multiplication, exponentiation, etc.

>> [-1,3,5] + [1,2,3]

**Plots**

In addition to computational tools, MATLAB has a many sets of graphical tools, the simplest being the ability to simply plot points from those new fangled vectors you’ve been creating. The plot function (in 2d) takes a vector *x* = [*x*1, …, *xn*] of x coordinates of points and a vector *y* = [*y*1, …, *yn*] of y coordinates of points and plots all the points (*x*1, *y*1),(*x*2, *y*2)….(*xn*, *yn*) joining them by straight lines. Note that the vectors have to be the same length! For example we can plot

>> x = [1,2,3,4,5]

>> y = [-3,4,6,2,1]

>> plot(x,y)

and this will plot the points (1,-3), (2,4), (3,6), (4,2) and (5,1) in order. What happens when you try to plot the points given by the vectors

>> x = [1,1,1,1,1]

>> y = [-3,4,6,2,1]

>> plot(x,y

Why do you think this happens?

**Exercise:**

Using the plot command, plot a square

**Exercise:**

Using the plot command, plot a star or a smiley face. Or if you to baaad for that a skull and cross bones.

**Plotting Functions**

The last step here is to plot functions over a desirable range. Let say we wanted to plot sin(*x*) for 0 ≤ *x* ≤ 10. A graphing calculator pretends that it can do this in a “continuous” fashion but MATLAB does not, you have to give it a list of pairs of points (*x*1, sin(*x*1)), (*x*2, sin(*x*2)), … and it will plot those. Lets try having MATLAB generate a vector of numbers between 0 and 10, take sine of them and plot that:

>> x = 0:10 % Sets x to the vector [1,2,3,4,5,6,7,8,9,10]

>> y = sin(x) % Sets y to the vector [sin(1),sin(2),sin(3),...]

>> plot(x,y) % Plot the points (1,sin(1)), (2,sin(2)),...

A bit jagged isn’t it. That because we only have 10 data point, ie x = [1,2,3,4,5,6,7,8,9,10]. How do we make this better? Well we could get 100 data point by only incrementing each number by 1/10 instead of by 1. Think about how you’d do this for a second before reading on for the answer.

My solution would be to write

>> x = 0:.1:10 % Sets x to the vector [1,1.1,1.2,1.3,...]

>> y = sin(x) % Sets y to the vector [sin(1),sin(1.1),sin(1.2),...]

>> plot(x,y) % Plot the points (1,sin(1)), (1.1,sin(1.1)),...

Much better.

**Exercise:**

The plot function has many different modifiers associated with it. The easiest way to find out about them is through the MATLAB documentation (or google). To find the documentation for the plot command type doc plot. Using the documentation, plot *y* = *sin*(*x*) from −*π* to *π* with a blue line, label the *x*-axis “Time” and the *y*-axis “Height”

**Exercise:**

Typing the function hold on allows you to draw multiple graphs to the same plot. To stop drawing to the same plot use the command hold off. Plot *y* = log*x* and *y* = *ex* on the same plot from 1 to 2, one in green one in blue.

**Exercise:**

Typing the function hold on allows you to draw multiple graphs to the same plot. To stop drawing to the same plot use the command hold off. Plot *y* = log*x* and *y* = *ex* on the same plot from 1 to 2, one in green one in blue.

**Exercise:**

The function axis allows you to specify the scale of the axes. Plot *y* = cos*x* for *x* from -*π* to *π* and on the same graph plot *y* = *x*2. Then, rescale the graph so that the *y*-axis goes from -2 to 2.

**Functions**

Functions are very different in MATLAB than in math. In MATLAB, a function is any bit of code that runs as a discrete chunk. For example, the following function takes no inputs and has no output but when we run it plots *y* = *x*2 for *x* ∈ [0, 10]:

function [args] = paraPlot

x = 1:10;

plot(x,x.^2)

end

Type the code above into the text editor and save the file as paraPlot.m. Then, when we type paraPlot into the command line, the code runs and plots *y* = *x*2.

Running the same code again and again is all good but the real power of functions is the ability to run the same code but on different variables. For example, what if we want to plot*y* = *x*2 for some range other than 1 to 10? Consider the following change to the code:

function [y] = paraPlot(x)

y=x.^2;

plot(x,y)

end

This code takes a variable x, that we assume to be a vector. It squares x and saves the value as y. It then plots x vs y and prints y to the command line. In general, we write functions as

function [out1,out2,...] = paraPlot(in1,in2,in3,...)

where in1, in2,... are the inputs and out1, out2,... are what the function returns to us after running. When you run a function and it gets to the bottom of the code (where the end is) it “returns” whatever has been assigned to the variables out1, out2,.... Of course you don’t have to call them out1, out2,..., you can call them sally, frank,... or whatever else you want although labeling them for clarity is usually recommended.

**Exercise:**

Lets create a function that factorizes the polynomial *ax*2 + *bx* + *c* using the Pythagorean theorem. The function should take three inputs, a, b and c and return the roots of the polynomial. Since we have three inputs and two outputs the first line of code will be

function [root1,root2] = pythag(a,b,c)

More complicated functions, like polynomials.

**Loops**

So now that we have a place to store our work lets begin making MATLAB work for us. The first thing we want to do is tell MATLAB to do something a number of times and for this we have loops. Lets look at the following code:

for i = 1:10 % Set i to each number from 1 to 10 sequentially

1 + i % Compute i + 1

end % Return to the top of the loop and set i to the next number

Note, everything after the % sign is a comment and will not be by the computer! Run this code and look at your command line to see the output. What this code says is

• Set i = 1

• Compute 1 + i

• Return to the beginning of the loop

• Set i = 2

• Compute 1 + i

• Return to the beginning of the loop

• ⋮

• Set i = 10

• Compute 1 + i

• Continue running the rest of the program (of which there none).

Alright, that’s very nice but what else can we do? Well we can write a program to say sum all the numbers from 1 to 100:

total = 0

for i = 1:100

total = total + i

end

This program starts by creating a new variable total which will hold the current amount of our sum and setting it to 0. It then runs the loop: *Set i = 1, set total = total + i*, now total currently is 0 and i currently is 1 so this line sets total = 0 + 1. Now we reach end so we return to the top of the loop: *Set i = 2, set total = total + i*. This time total currently is 1 and i currently is 2 so this line sets total = 1 + 2. What will it be on the next step?

How would you use this code to sum the numbers from 7 to 50? How could we change this program so that it computed to factorial of 100?

*Note:* At this point (or some point soon) you may be wondering if there’s any way to make the program not dump 100’s of line of numbers to the command line every time you run it. The answer is yes, if you ever don’t want a program to print a calculation simply end your line with a semicolon “;” for example

total = total + i; % <— With A Semicolon!

will suppress almost all of the program’s output. So how do you get it to print the final answer? Well you can always add a last line that’s just total, you can find the variables you’re interested in in the workspace and read them off there, or you can find another way to represent the output.

**Keeping track of numbers in arrays**

Some times seeing just the result isn’t good enough, we want to actually store the information we compute at each step so we can analyze it later. We will do this by building an array.

We can access elements of an array through indexing. For example if you type

>> x = -3:3

>> x(4)

then x(4) returns the fourth element of the array (in this case, 0). Similarly typing x(3) = 7 sets the third element of the array to 7. We can use this and loops to build more complicated arrays.

Let us extend the code above so that instead of just giving us the answer it builds an array of all the sums less than 100:

total = 1 % Set ‘total’ to 1

for i = 1:100 % We let i run from 1 to 100

total(i) = total(i-1) + i; % Take previous total, add ‘i’ to it

end % Back to the top, pick the next ‘i’ plot(total)

What does this code do? Well, MATLAB treats every variable as a one element array so just by setting total = 0 we’ve really set the first element total, that is total(1) to 0. Next we loop through starting at i = 2 (we’ve already set total(1)) and compute the next number in the sequence from the previous one. In this case, we’re just adding *i* to each previous number so for example we have total(2) = total(1) + 2. On the next run through we’ll have total(3) = total(2) + 3, etc. After this program is done we can plot all the sums we have computed, or look at them in the command line.

Plotting it, it actually looks pretty smooth. I wonder if there’s a formula behind it?

**Exercises**

**The Fibonacci Sequences**

1. Modify the code from the second example to compute the factorial of 100, ie

100! =1 ⋅ 2 ⋅ 3 ⋅ … ⋅ 99 ⋅ 100.

2. The Fibonacci sequence is one of the more common sequences used to model nature due to the way it describes growth. The sequence is given by the formula

…

For example the third number is *x*3 = 1 + 1 = 2 and the fourth number is *x*4 = 2 + 1 = 3. Can you write a for loop that computes the 20’th Fibonacci number? Note, you might need more than one extra variable.

3. If you’ve finished the above, try plotting each of the Fibonacci numbers. There are many of ways to do this but we might suggest looking at the function hold on or storing all the Fibonacci numbers in an array and simply plot the array at the end.

4. Plot the ratios of the successive Fibonacci numbers, ie plot (*x*1/*x*2, *x*2/*x*3, …). What do you find?

**Plots**

One use of loops is to plot multiple pieces of data on the same plot. For example the following code plots a simple line 10 times but changes the end points:

for t = 1:10

plot([1,0],[1,t])

hold on

end

Note the hold on! This lets us draw to the same graph. Try deleting it and see what happens. Play around with this code a bit see if you can make sense of it/break it, make it do something interesting.

1. Can you modify to for loop to draw the lines (0,10) to (1,0), (0,9) to (2,0),..., (0,1) to (10,0)? Can you do the same thing but start with a line (0,100) to (1,0)?

2. Recall that to plot a function we specify an *x* range with by x = a:b for example, and then write plot(x,f(x)) where f(x) is the function we’re trying to plot. We can get an idea of how function change under parameters by plotting multiple functions on the same graph. For instance, can you plot sin(*x* + *t*) for *t* = 1 to 10 on the same graph?

**If Statements**

Alright, so now that we’ve gotten our hands dirty with loops we’ll introduce the other basic ingredient of computer programing: if-then statements (or how to make a computer make decisions for you). An if then statement allows you to have your program change while its running, with no input from you. Lets look at an example:

m = 50

n = 70

if(m+n < 100) % If it’s true that m+n<100, do the following

y = m\*n

else % If it’s NOT true that m+n<100, do the following

y = (m + n)/2

end

What is the value of *y* here? What would the value of *y* be if *m* = 20 and *n* = 70 ? What about if *m* = *n* = 50?

**Example: The Fibonacci Sequence**

The real power of if statements, or **conditional statements** as they’re often called, is when they’re used in conjunction with loops and other programing structure. For example, let us say we wanted to know the following:

*Recall that the Fibonacci sequence is given by* *x*1 = 1, *x*2 = 1, *xn* = *xn*− 1 + *xn*− 2. *What is the first digit* *xN* *of the Fibonacci sequence that is greater than 1,000,000 and what is N for this element?*

How do we think about this problem? Well, I could just start listing numbers and wait until one looked big enough then check and count all the ones before that but that sounds time consuming. Of course I could have MATLAB do all that for me but how? Whenever you’re given a problem it’s important to break it up into pieces. Here I can see at least two pieces:

1. Find the first digit of the Fibonacci sequence that is greater than 1,000,000.

2. Find out which element of the sequence it is.

We can see that we’re not going to be able to do the second part without doing th first so lets start with finding the number. Recall the code to generate elements of the Fibonacci sequence:

x = 1; % Set the 1’st element of the array x to 1

x(2) = 1; % Set the 2’nd element of the array x to 2

for i=3:1000

x(i) = x(i-1) + x(i-2); % Set the i’th element of the array x to the sum of the previous two elements

end

This will compute the first thousand Fibonacci numbers. Lets run this code. Well it looks like that was too many. It would be nice if we could tell the program to stop when it’s gotten out number. That’s where if statements come in. Lets use an if statement to check each element as we compute it and see if it’s bigger than 1,000,000:

*First Attempt*

x = 1; % Set the 1’st element of the array x to 1

x(2) = 1; % Set the 2’nd element of the array x to 2

for i=3:1000

x(i) = x(i-1) + x(i-2); % Set the i’th element of the array x to the sum of the previous two elements

if x(i) > 10000000 % If the i’th element of x is > 1000000

x(i) % Print the i’th element of x

end

end

Well this sort of works, it finds the first number bigger than 1000000 but then spits out other numbers larger than 1000000. If we want it to stop we can use the command break:

**break** - immediately ends a loop, continue with the next line of code after the loop

Lets add break in the if statement above so that the first time it finds a number greater than 1,000,000 it will stop computing more elements of the Fibonacci sequence:

*Second Attempt*

x = 1; % Set the 1’st element of the array x to 1

x(2) = 1; % Set the 2’nd element of the array x to 2

for i=3:1000

x(i) = x(i-1) + x(i-2); % Set the i’th element of the array x to the sum of the previous two elements

if x(i) > 10000000 % If the i’th element of x is > 1000000

x(i) % Print the i’th element of x

break

end

end

Now we see that our program outputs just one number. A couple things to think about: (1) do we know that this is actually the correct number? How could we check? (2) How do we determine which element of the sequence it is (ie the 1’st, 2’nd, 500’th, etc).

**Logical Expressions**

The logical expressions that can be used in MATLAB

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| & | And |  | | | Or |
| ~ | Not |  | > | Greater than |
| < | Less than |  | >= | Greater than or equal to |
| <= | Less than or equal to |  | == | Equal to |

For example the following code sets x to 10 and y to 33 and then evaluates some logical expressions. Does it evaluate to true (1) or false (0) for each expression? See if you can figure it out and then enter the code into the command line to check your answers

x = 10

y = 33

(x^2>6) | (y >= 100)

(x^2>6) & (y >= 100)

~(x\*y == y)

~(x - y>23 )|(~(x+y<50))

**Exercise:**

Modify the code of your quadratic formula calculator to do the following: Take three variables a, b, c, corresponding to *f*(*x*)=*ax*2 + *bx* + *c*. Plot *f*(*x*) over some interval. If *f*(*x*)=0 has real solutions plot them on the same plot as *f*(*x*). Return the solution to the command line.

**Exercise:**

Make a function that takes a variable n and plots an *n*-gone.

**Exercise:**

Make a function that takes a vector of *y*-values and uses a for loop to compare each point with its neighbors to find all the max’s, mins and saddle points. Plot the vectors of *y* values along with each of these.

**Exercise:**

MATLAB can of course do the above for you. See if you can figure out how to do it in one line of code using built in MATLAB functions.

**Projects**

Project guidelines: In this project you will...

Once you have either finished the project or gotten as far as you reasonably can in 4 hours you should compile a presentation about the project. This presentation should be short, ~5 min. The presentation should follow the format

• What was the project about? *(Eg. The project was about counting numbers)*

• What did you actually do? *(Eg. We tried to create a program that counted all the numbers, we were not successful but we did manage to count all the numbers up 100)*

• How did you do it, explain your code. *(Eg. We used a for loop to count to the biggest number we could think of)*

• What problems did you run into and how did you solve them. *(Eg. We couldn’t count all the numbers so we redefined the scope of the project to only count to 100. We do not think that the original project is not technically feasible, due to a slight generalization of Euclid’s Theorem.)*

You can either use a power point to show your results or use MATLAB to run and explain the code yourself. Feel free to ask to mentors for help, especially with understanding the technical vocabulary, but all the word should be yours.

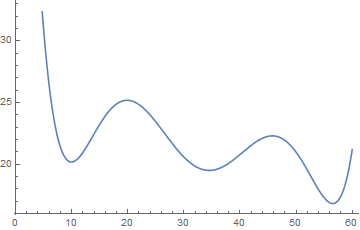
Always remember to save your code and email it to all the group members!

**Project (polynomials) - Min, Max**

In this project we’re going to build a MATLAB program that analyses a polynomial and finds all of its maxima and minima points and zeros. This is one of the most important problems in *applied mathematics*: applied mathematics is the process of coming up with equations that simulate real world phenomena and using those equations to answer questions or make predictions about the world. Some of the most common questions are of the form “How can I maximize the output of this factory?” or “How can I minimize the amount of time it takes to get from school to work?”

Let us say that you take the train to school every morning. After some extensive “research” you realize that with the various buses and trains if you leave *t* minutes after 7 o’clock in the morning it takes

minutes to get to school. You don’t care much about when you get there you just want the shortest trip. Looking at the graph below it’s easy to pick out the longest and shortest trips:



We want to automate this process so that MATLAB can figure out for you where all the maximums and minimums are. In this project you will make a program that takes a vectors of values *y* = *f*(*x*), plots it, and plots all of the maxima, minima and zeros of the function.

1. Make a function that takes in vector of *x* values and returns *y* = *f*(*x*). Plot *y* = *f*(*x*).

2. Discuss with your team: what makes a point a maximum or a minimum? When you have an answer tell one of the mentors.

3. The criteria you found above should be of the form “a minimum is lower then all the points around it.” We want to find all points such that is condition is true. To do this we need to compare each point to it’s neighbors as see which is bigger. To compare things we must use if then statements:

if (something is true)

do something

else

do something else

end

Write a function isMax that takes three numbers xleft, x, xright and returns “true” if xleft<x and x>xright and “false” otherwise. Since there are two conditions you will probably need two if statements. Protip: you can put if statements inside if statements but putting them in the “do something” code.

4. Write a function isMin that tests three points xleft, x, xright and returns “true” if x is the minimum.

5. Write a function plotMaxMin that takes vectors x and y and plots *y* = *f*(*x*). Then, use a loop to test all triplets of points *yn*− 1, *yn*, *yn*+ 1 and find all the maxes and mins. Plot *y* = *f*(*x*) and then plot all the maxes and mins.

6. The command text(x1,y1,’some text’) allows you to add labels to to a plot. Label each of the maxes and mins with their coordinates

**Extra**

7. Add to the plot all the points where *f*(*x*)=0.

8. Add to the plot the absolute max and absolute min (over a range of your choice)

**Project (calculus) - The rate of change**

At its most basic, calculus is the study of how the velocity (speed + direction) of an object relates to its position. The principles of calculus can then be extended to anything that changes over time, from populations of people to the stock markets to levels of chemicals in your body. In this project we’re going to explore one simple aspect of this relationship: given a function that tells us the position of a ball at a certain time, how can we find the speed of the ball at that time.

**Part 1**

(1) Imagine that we push a ball and it roles away from us. The position of the ball *t* seconds after we push it is given by

What kind of function is this? Use MATLAB to graph this function for the first 10 seconds, ie from *t* = 0 to *t* = 10.

(2) Speed is the distance something can go over a certain period of time. For example, if the ball is traveling 10 miles per hour in 1 hour it will go 10 miles. If *t* in part (1) measures seconds and *p*(*t*) is feet then how many feet does the ball travel in 10 seconds? At what speed is the ball going?

(3) In general, the average speed of a function between two points is just the slope of the function between those points:

*Avg. Speed between times and : .*

Check that this formula gives you the same answer for the speed you got in part (2) by plugging in the start and end points. For the function *p*(*t*), does it the average speed change depending on which points you use at the start and end point? If not, we say that the speed is constant. When you have an answer call over a mentor and check it with them.

(4) What is the average speed (average rate of change) for *p*(*t*)=10*t* − 4 Use both a plot and the formula to find the answer.

(5) If the graph of a function is a line, what is the relationship between the slope of the line and the speed?

**Part 2**

In the examples in part 1 the speed was always constant. This is what we’d expect for a thought experiment but in the real world things change speed all the time. Lets think about a slightly more complicated example:

Imagine we throw a ball up in the air. At time *t* the ball is at position *p*(*t*) where

(1) What kind of function is *p*(*t*)? Use MATLAB to plot the graph of this function from *t* = 0 (when the ball is thrown) until the time at which the ball hits the ground.

(2) We want to know how fast the ball is going up (or down) at various times. Looking at the plot, we can see that the ball goes up the fastest right when we release it from our hand and slows down until it stops at the maximum (*t* = 1) and then turns around and starts falling. It falls faster and faster until it hits the ground at *t* = 2. Using the formula above:

What is the average speed of the ball between the time you throw it at *t* = 0 and its maximum at *t* = 1?

What is the average speed of the ball between the maximum at *t* = 1 and when it hits the ground at *t* = 2?

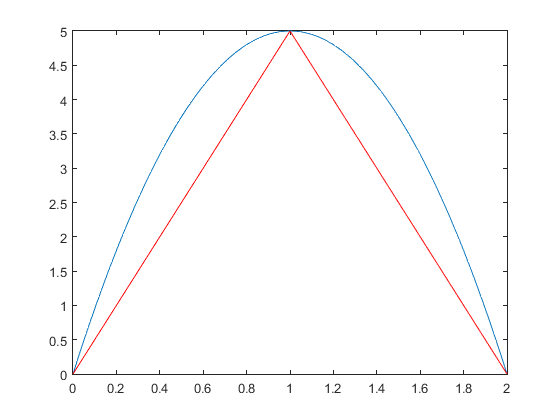
(3) If the speed the ball is moving is constant, the graph of it’s position should be a straight line. Why is this? Ask a mentor if you do not understand this point.

If we want to approximate the balls motion with two straight lines, one going from *t* = 0 to *t* = 1 and one from *t* = 1 to *t* = 2. To do this we would type

t = [0,1,2]

y = -5\*t.^2 + 10 \*t

plot(t,y)



Activity 3: This plot contains the arc of the ball and two linear graphs we use to find the balls average speed over an interval.

Put the plot of the whole parabola on the same graph as the parabola.

(4) Plot the average rate of change for each of these intervals (what we computed in Question (2)). We should get a plot with two points, one positive, one negative, connected by a line.

(5) Instead of estimating our ball throw with 3 points, *t* = 0, 1, 2 we could estimate with 5 points, *t* = 0, .5, 1, 1.5, 2. Plot the parabola with the with 5 point estimate. Compute the average speed between each two points. Plot these average speeds. What do you find?

(6) We want to be able to estimate the average speed even more precisely. We could cut the parabola up into even more pieces but that’s time consuming for us. Fortunately, it’s not for a computer! Use x=a:b:c to cut the parabola up into 10 intervals (pieces) and plot them.

Now, we want to compute the average speed of each over each of the 10 intervals. We need to compute (y(2)-y(1))/(t(2)-t(1)), (y(3)-y(2))/(t(3)-t(2))... all the way up to 10. Use a for loop to do this and save each element in a vector called speed.

(7) Plot the vector speed, what do you notice?

(9) Change your code so that you can estimate using 100 intervals. Or 1000 intervals.

(10) Let *p*(*t*)=*t*3 − *t*. Approximate the speed off this function for *t* = −2 to *t* = 2 for 100 intervals.

**Extra** It would be nice to see how the average speed matched up with different parts of the graph of the position. We can put two different graphs on the screen at once by using the command subplot(m,n,p) creates a plot in the p’th position of an m by n grid. Try creating a few subplots in different positions. For example, the code:

t = 0:5

y1 = -5\*t.^2 + 10 \*t

y2 = sin(t)

subplot(1,2,1)

plot(x,y1)

subplot(1,2,2)

plot(x,y1)

makes two plots, the top one being of a parabola and the bottom one being of a sin function.

Now, use subplot to make two plots, one which plots the function and one of which plots the average speed (you can pick the number of intervals you want to use).

**Project (calculus) - Area under a curve**

One of the most basic topics in calculus is trying to find the area under a curve. In calculus, we relate this interesting geometric question to a wide variety of topics but here we want to explore the question itself.

(1) Lets start simple: Consider the graph of *y* = *x*. What is the area between the graph from *x* = 0 to *x* = 1 and the *x*-axis? What about between *x* = 0 and *x* = 10? Use MATLAB to plot *y* = *x* from 0 to 10 and use trig/geometry to figure out the area.

(2) Now, what about the area between the *x*-axis and the graph *y* = 2*x* − 6 with *x* from 0 to 2?

(3) Plot the graph *y* = −5*x*2 + 10*x* for *x* = 0 to *x* = 2. We would like to figure out the area between the *x*-axis and the parabola, but the parabola is curved and we don’t have any formula! For a general function, there might not even be a formula. What can we do? One option is that we can estimate the area using shapes we do know the area of and then try to make our estimates better and better.

Estimate the area under the curve by using the smallest box that fits the entire area. What do you get?

(4) What if we used two right triangles to estimate the area? Estimate the area by cutting it in half and using a right triangle to estimate each half. In MATLAB, we can visualize this with the code

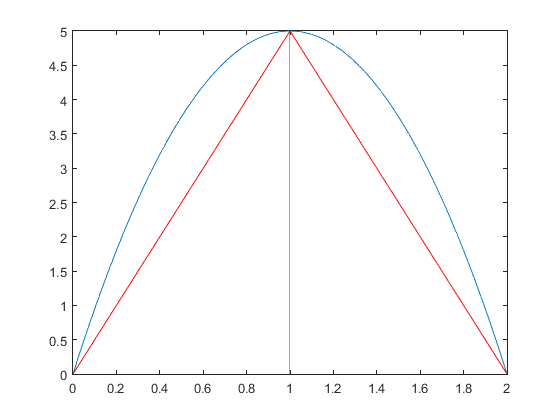
x = [0,1,2]

y = -5\*x.^2 + 10 \*x

plot(x,y)

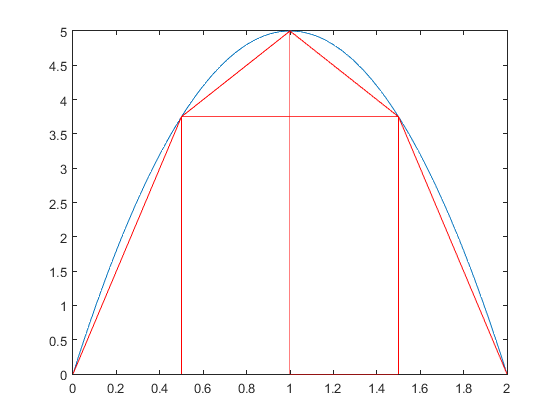
hold on

plot([1,1],[0,5]) % Draw a line down the middle of the picture



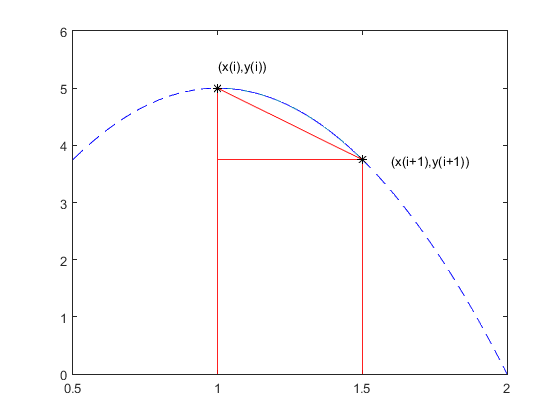
Activity 4: This plot contains the parabola and the two triangles we will use to estimate its area.

(5) In general, its always simple to estimate the area of boxes and right triangles. Cut the area under the graph into 4 pieces and estimate each one by a right tangle on top of a box like in the picture. What do you get? How would you visualize this with MATLAB? Use the code above as an example. Once you’ve figured out the answer check it with a mentor.



Activity 5: This plot contains the parabola, we cut it into four pieces and estimate each one by a box with a triangle on top.

(6) We want to be able to do this for many, many more intervals. By hand it would quickly become impossible but for a computer it’s easy, all we need to do is figure out a formula for each of the trapezoids we’re using to estimate each interval. Lets say that the corners of the trapezoids are at coordinates (x(i),y(i)) and (x(i+1),y(i+1)). What is the area of the trapezoid? Check with a mentor before continuing.



Activity 6: For coordinates (x(i),y(i)) and (x(i+1),y(i+1)), what is the area of the quadrilateral?

(7) Create a vector x = [0, .5, 1, 1.5, 2] containing the end points of the four intervals. Using the formula you derived in (6) and a for loop, add up the area above each interval. Is the total area the same as what you computed by hand?

(8) Change the code in (7) so that there are 8 intervals instead of 4. What do you get for the total area? Does this make sense? Check with a mentor before continuing.

(9) Change the code in (7) so that there are 100, or 1000 intervals. What do you get?

(10) Estimate the area under the function *y* = 4 − *x*3 + *x*2 from *x* = −1 to 1 for 1000 intervals. What do you get? What do you think the real answer is?

**Project (trigonometry) - Waves and sound**

For this project we will explore how MATLAB can generate, record and simulate sound. To start with lets nail down the terminology we will use in this project:

• *t* - time in seconds

• *f* - frequency, ie number of waves per second (units of hertz, Hz)

• *A* - amplitude, how tall the waves are and how loud the sound is

A sound wave is a wave of compressed air. For a pure tone, ie a sine wave, the pressure hitting our ears at time *t* will be given by *w*(*t*)=*A* \* sin(2*πft*) compare the formula with the picture below.

(1) The following code plots a wave with frequency f, amplitude A, that lasts for T seconds and has Fs points per second:

A = 10; % Amplitude

f = 50; % Frequency

T = 1; % Length of sample

Fs = 1000; % Sample Frequency

t = 0:1/Fs:T;

wave = A\*sin(2\*pi\*f\*t);

plot(t,wave)

What happens as you change the variables f, A, T and Fs? You can use hold on to draw multiple waves to the same plot.

(2) MATLAB can play the wave we’ve created as a musical note, just type sound(wave, Fs). Create a wave with *A* = 10, *T* = 2, *Fs* = 1000 and *f* = 200 and play it. Try this for different waves. How does changing f, A, T and Fs effect the sound?

(3) Most sounds that exist in nature are not pure tone but composites of pure tones, that is pure tones added together. Create two waves, wave1 and wave2, one of frequency 300 and one of frequency 600. Play them both separately and then play wave1 + wave2. What do you find?

(4) Keeping wave1 fixed at the frequency 300 generate wave2 waves of the frequencies *f* = 350, 400, 500, 600, 700, 900, 100. Listen to wave1 + wave2. Which do you think sound best together? Do you notice anything about the frequencies that sound good together?

**Aside:** We can put two different graphs on the screen at once by using the command subplot(m,n,p) creates a plot in the p’th position of an m by n grid. Try creating a few subplots in different positions. For example, the code:

t = 0:5

y1 = -5\*t.^2 + 10 \*t

y2 = sin(t)

subplot(1,2,1)

plot(x,y1)

subplot(1,2,2)

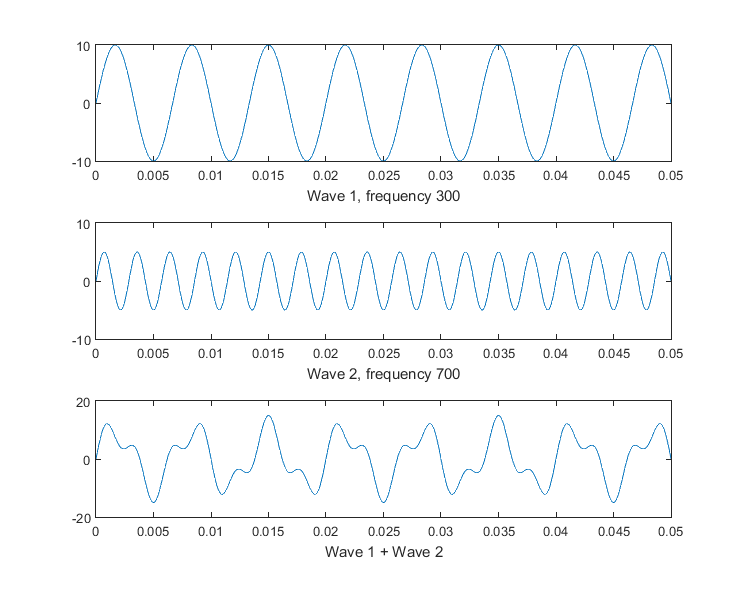
plot(x,y1)

makes two plots, the top one being of a parabola and the bottom one being of a sin function.

(5) Using either sub figures or hold on plot wave1, wave2 and wave1+wave2 on the same screen.

(6) Make a function that takes two frequencies, plots waves of those frequencies and plays the sound they make. It may be helpful to use the function pause(n) to make the program pause for n seconds to let each sound play.

(7) Extend your function from (6) to also plot the sum of the two waves and play the resulting sound. Once you have this working, talk to a mentor about your code.



Here’s an example of how the output of your function might look.

**Part 2**

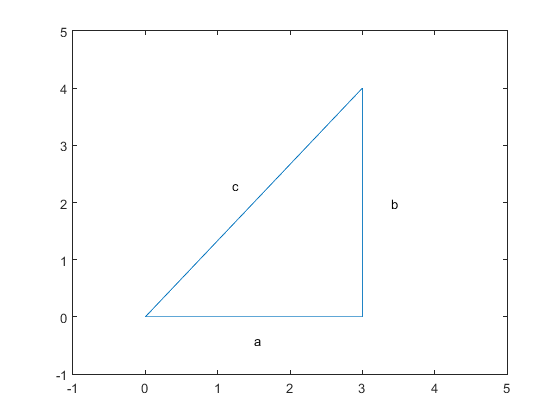
(1) Now, assume that we have a sound wave, how do we find out what frequency it is? Well, we could look at the graph and count but that’s slow. The faster way is to use what’s called a Fast Fourier Transform (FFT). The math behind a Fourier transform is a little too complicated for this class (it requires already knowing calculus) but what the function does is quite simple, put in a wave and it returns the frequency:

**Project (trigonometry) - Triangle Calculator**

In this project we will use MATLAB to make several function that will compute everything we’d ever want to know about triangles for us. First, I’ll do an example, then it will be up to you to figure out how to program the remaining functions.

**Part 1: Sides to angles**

We want to make a function that takes as inputs the opposite and adjacent sides of a triangle and computes the length of the hypotenuses, and all the angles. It will then draw the triangle and label each of the sides. Lets say we have a right triangle with sides *a*, *b* and *c* like in the picture



A right triangle with short sides *a* and *b* and hypotenuse *c*.

**1. Calculating the length of the hypotenuse**.

We want to make a function that takes as inputs the lengths of *a* and *b* and returns the length of the hypotenuse *c*. We know how to calculate the hypotenuse by hand:

So let us start with the following code:

function [ c ] = triCalc(a, b)

c = sqrt(a^2 + b^2);

end

Type this code into the editor and save it as **triCalc.m**. This code defines a function called triCalc that takes two inputs, a and b and returns whatever is in the variable c when we hit end. On the second line, it calculates the square root of a^2 + b^2 using the sqrt() function and saves it to the variable c. The last line is end which just tells the function to print whatever is in the variable c to the command line. Go ahead and try it out for some values you know, you should get

>> triCalc(3,4)

ans = 5

for example.

**2. Adding a plot.**

This is a good start, but now we also want to draw the triangle. We can do this with a plot function we just need to know where each of the corners is. Looking at the picture we can see that starting to left and going counterclockwise the first corner is (0,0), the second is (a,0), the third is (a,b) and then we come back to (0,0). Recall that for plot we need two vectors, x containing all the *x*-values and y containing all the *y* values. We can write x = [0, a, a, 0] for the *x* vector, what do we type for the y vector?

When you know add the code to the triCalc function and you should be able to plot triangles of different sizes! Although always look at your axes because MATLAB will try to be “smart” can scale the axes so that all the triangles look the same.

**3. Fixing the axes.**

You’ve probably noticed that your plot doesn’t appear to contain a triangle, only a line down middle. This is because MATLAB zooms all the way in as far as it can after it draws the triangle. To fix this we can use the axis command to zoom back out. If you don’t remember the axis command it’s in the first handout, or you can always type help axis into MATLAB. We write

axis([xmin, xmax, ymin, ymax])

Where you should replace xmin, xmax, ymin, ymax with the minimum and maximum *x*-values and minimum and maximum *y* values respectively. For example, we could write

axis([-1, 10, -1, 10])

to scale the picture to go from -1 to 10 for both *x* and *y*. This is fine but for very large triangles we won’t be able to see everything! Another option is that we can put the variables a and b into axis, for example

axis([-1, a+1, -1, b+1])

This changes the scale as we put in different a and b. Try putting one of these lines of code in to your function after the plot code. Which do you like better? As a group, try out different values of a and b and see which you like better.

**4. Adding labels.**

Okay, we’ve got a pretty nice picture now but it would be nice to be able to label the sides with their lengths, especially the hypotenuse. We can add text to a plot with the command text(x,y,’your text here’) where (x, y) is the point you want the text to appear. For example, we could add the label

text(a/2,-.4,’a’);

to label the bottom line ’a’. Notice that we’ve put the *x*-coordinate at *a*/2 to place it half way down the bottom edge. See if you can add labels for the sides *b* and *c*.

**5. Adding numbers to labels.**

Now, as mentioned above we want to add the actual lengths to the plot so that we can easily read them off. But the numbers are stored in the variables a, b and c and as you can try yourself putting these in gives us an error. What the error is trying to tell us is that the variables a, b and c hold numbers, not strings (strings just means strings of characters, so any word, sentence or complete nonsense). There’s a command to convert numbers to strings, namely the num2str() command. Then we can write

text(a/2,-.4,num2str(a));

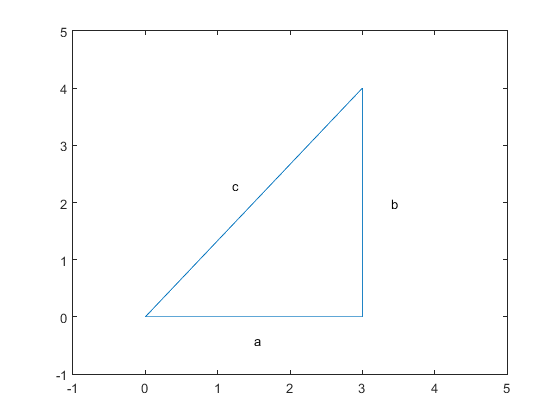
to label the bottom with the length of the bottom. Label the other sides with their lengths as well. As a final note, you can combine numbers and strings by putting them into a vector together. For example, if we write

text(a/2,-.4,[’a = ’, num2str(a)]);

MATLAB will add the label a = 4 or whatever the length of side *a* is.

**6. Calculating angles.**

At this point, we know almost everything we’d ever want to know about the triangle. As a final test, let us calculate all of the angles and label them. Now, one we already know, that’s the right angle in the corner. The others we can figure out from the inverse sine and cosine functions. How do we do this? Remember that



That means that, looking at the picture, we can write sin(*ang*1)=*b*/*c*. Since we know *b* and *c* we can use arcsine to write

ang1 =

In MATLAB, the arcsine function is asine. Similarly we have acos and atan. Note, all of these functions are in radians, for degrees use asined, acosd and atand. So MATLAB can compute the angle of ang1 for us by using the code

ang1 = asin(b/c);

How would you compute ang2? Add code to your function to compute both ang1 and ang2 and label the angles with them

**Part 2: Angles and side**

With only one angle and one side you can compute all the other information about a right triangle. That is to say, given only ang1 and *b* as in the picture we can use to compute the hypotenuse and other trig identities to compute the length of the final side and the other angle. Make a function that takes two inputs, ang1 and b and computes a, c and ang2 and draws the triangle. You should be able to use most of your code from the last part.

**Part 3: Challenge**

Write a function that takes three numbers, a, b, c, as inputs and draws a triangle with side lengths a, b, c. This is kind of a difficult conceptual problem so think about how you would do this in real life before you begin coding. Suggestions are available upon request.

Creative Commons License

This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](http://creativecommons.org/licenses/by-sa/4.0/).